

# Hardware Parallelism: Are Operating Systems Ready? (Case Studies in Mis-Scheduling)

Eitan Frachtenberg

Modeling, Algorithms, and Informatics Group  
Computer and Computational Sciences Division  
Los Alamos National Laboratory  
eitanf@lanl.gov

Yoav Etsion

School of Computer Science and Engineering  
The Hebrew University  
Jerusalem, Israel 91904  
etsman@cs.huji.ac.il

## Abstract

Commodity parallel computers are no longer a technology predicted for some indistinct future: they are becoming ubiquitous. Commodity processors with parallel program execution abilities are produced by every major chip manufacturer. In the absence of significant advances in clock speed, chip-multiprocessors (CMP) and symmetric multithreading (SMT) are the modern workhorses that keep Moore's Law still relevant.

On the software side, we are starting to observe the adaptation of some codes to the new commodity parallel hardware. While in the past, only complex professional codes ran on parallel computers, the commoditization of parallel computers is opening the door for many desktop applications to benefit from parallelization. We expect this software trend to continue as the only apparent way of obtaining additional performance from the hardware will be through parallelization.

Based on the premise that the average desktop workload is growing more parallel and complex, this paper asks the question: Are current desktop operating systems appropriate for these trends? Specifically, we are interested in parallel process scheduling, which has been a topic of significant study in the supercomputing community, but so far little of this research has trickled to the desktop operating system. This paper checks the adequacy of process scheduler in prevalent general purpose operating system to common parallel programming paradigms, presenting experimental results a variety of parallel architectures.

Our main finding is that neither commodity desktop schedulers nor parallel schedulers do very well in all cases. Parallel desktops with complex architectures and workloads may require a unified approach to scheduling that stems from an understanding of the requirements of all process classes and their mixes. as well the abilities of the underlying architecture.

## 1 Introduction

Computer performance has improved at an exponential rate since the introduction of the first microprocessor. The effect of this growth is probably most visible in the way commodity computing has permeated almost every aspect of our daily lives. As the arbiter between hardware and software, the operating system (OS) has had to keep pace with advances in both software and hardware to permit this growth. Subsequently, we have witnessed marked improvements in the way OSs handle input and output devices, user interfaces, networking, and interoperability. Process scheduling, however, has seen little progress since the introduction of timesharing with CTSS. While simple adjustments, tuning, and handling of special cases was incrementally added to all commodity schedulers, the basic scheduling principles have remained practically unchanged. This stagnation was largely tolerated until recently, since the steady rate of performance increases masked most scheduler inefficiencies. In this paper we ask whether current trends in hardware and software will require a shift to new scheduling policies that are better-suited for parallel hardware and richly complex workloads. An important goal of this paper is to motivate research into schedulers that take a variety of such cases into account, rather than just focusing on domain-specific scheduling problems. We support these claims with experimental data on the scheduling performance under forward-looking assumptions on commodity workloads and architectures.

To justify these assumptions, the rest of this section describes recent trends in hardware and software technologies. Section 2 elaborates on why these changes will render current commodity schedulers obsolete. To substantiate this claim, the main part of this paper (Section 3) demonstrates — using several case studies — some of the problems that we expect schedulers to perform poorly on with future hardware and software. Finally, we offer some concluding remarks and suggest a path forward in Section 4. This section describes our ongoing

and future work to address these problems by employing a more holistic approach to process scheduling that encompasses a better understanding of, and suitability to, the hardware and workload at hand.

## 1.1 Hardware Trends: The Move to Concurrency

The phenomenal increase of microprocessor performance has fueled a similarly explosive growth in commodity hardware and applications for more than two decades. However, the increase in single-processor performance is now showing signs of slowing down. Already supercomputers (with performance growth that outpaces that predicted by Moore’s Law) have shifted from single-processor architectures to parallel ones. Just as innovations in Formula One race cars are the precursor for many improvements in mainstream vehicles, innovations in high-end computers often portend advances in commodity architectures, as was the case for example with network and storage technologies.

To maintain high rates of performance growth, manufacturers are perforce turning to parallelism [22, 27, 31, 50]. Even single-processor and desktop computers are shifting toward parallelism, offered in SMT (though not always successfully [54]) and multicore (CMP) processors from Intel, AMD, IBM, Sun, and others [22, 31, 44, 47]. special scheduling requirements [7]. An example of such emerging architecture involving a relatively large number of special-purpose computing cores, is the Cell processor for media applications [27].

While parallelism itself will not solve all the problems that are restricting performance growth — such as energy budget, cooling capacity, and memory performance — we focus this paper on the inherent problems facing operating systems on the way towards adequate support for parallelism.

## 1.2 Software Trends: Increased Diversity

We assume in this paper that ubiquitous parallel hardware brings with it complex workloads and software will become increasingly more parallel and demanding. We posit that consequently, the scheduling requirements from the OS will grow more complex.

The wide availability of parallel hardware should conceivably provide an incentive for, and spark growth in, parallel programming. Although parallel programming has been successfully used for many years in specialized fields, most programmers have yet to make the challenging shift to parallel software design. This paradigm shift has been compared to the 1990s’ move to object-oriented design, also a technology that had existed for many years before becoming widely used [50]. The comparison permits us some optimism about this adoption. With increasing demand, powerful new parallel languages and programming environments could become widespread, as was the case with C++ and Java for object-oriented design.

Already a typical uniprocessor desktop with a multitasking OS runs multithreaded applications, motivated by considerations of resource overlapping, increased responsiveness, and modularity [17]. These applications range from the multithreaded Web browser to database and Web servers. The increasing parallelism in hardware has also revitalized research in compiler auto-parallelism [56, 6, 30, 26, 20, 23, 1]. The promise in this approach is that it leaves it to the compiler to identify potential parallelism in the code, and generate multi-threaded executables that can utilize the parallel hardware. This study area has already proven successful both in parallelizing loops [33, 1] and inter-procedural/task-level parallelization [26, 23].

History teaches us that commodity hardware innovations often trickle down to software development quickly. As such, the move to hardware parallelism in desktops will soon be expressed in the emergence novel parallel desktop application, which leads to the main focal point of this paper: are desktop operating systems ready for this change?

## 2 Problem Discussion: Parallel Workloads on Desktop Schedulers

The new parallelism trend in computer architectures challenges the general-purpose OS with workloads it wasn’t designed to handle. To explain on this claim, this section expands on the inadequacy of desktop schedulers to handle common parallel programming paradigms and workloads. These scenarios are later demonstrated with case studies in the next section.

Changes in desktop computer architecture promote changes in typical desktop workloads, thus motivating research in OS and process scheduling. The increasing clock speed trend of the past two decades has promoted multimedia computing, subsequently motivated research into this field. This in turn led to some novel scheduling policies for soft real-time and multimedia applications [36, 5, 10, 12, 24, 37, 38, 41]. Nevertheless, the performance of the commodity OS does not have a good track record of scaling up with the underlying hardware performance [39], and in particular, does not scale to multiprocessor workloads very well [42]. Parallel job scheduling has mostly been studied in the context of supercomputers and clusters [16], but rarely in the context of the commodity

desktop. We therefore predict that critical issues of parallel thread scheduling will surface on the desktop as the degree of parallelism increases in commodity machines in the near future. Even today’s simplest parallel machines, such as SM Ts or small SMPs and CMPs, already have difficulties with many applications and workload mixes [2, 7, 25]. Commodity schedulers are challenged at all levels of parallel execution, from the thread [7, 49], through the SMP [2, 53], the cluster [3, 15, 19, 52], all the way to supercomputers [29, 40, 53].

Current parallel programming paradigms are closely based on Flynn’s classic categorization of parallel processing [18, 58]:

- Single-Instruction-Multiple-Data (*SIMD*), also known as the *Workpile* programming model. In this model, the entire dataset is divided into smaller subsets that have to be processed symmetrically with no computational dependencies. The work can be divided into several identical threads, each handling a subset of the data from the general *workpile*, processing it, communicating the result and continuing on the next subset. During the computation there are no inter-thread dependencies, and the overall computation completes after all data subsets have been processed by *any* thread.
- Multiple-Instruction-Single-Data (*MISD*), also known as the *Systolic* or *Pipelined* programming model. In this model, the computation (rather than the dataset) is divided with each thread performing a subcomputation on the dataset, then pushing it off to the next subcomputation-thread, and freeing itself to process the next dataset. Each thread depends on the completion of the previous subcomputation-thread, and the computation is completed when the *entire* dataset is processed by *all* threads in turn.
- Multiple-Instruction-Multiple-Data (*MIMD*) also known as the *Bulk-Synchronous* Parallel (BSP) programming model. In this model, both the dataset and the computation are divided into subsets, with each thread processing *all* the data subsets. The computation itself is phased, such that each thread performs its subcomputation on a subset of the data during a global computational phase. When all threads complete their subcomputations they enter a communication phase in which boundaries are exchanged between threads. The overall computation completes when *each* data subset have been processed by *each* applicable thread.

The predominant approach to multiprocessing in general-purpose OSs is to treat each processing element as an independent entity—processes/threads are migrated between processing elements in an attempt to balance cache affinity needs with CPU load imbalance [34, 9, 35, 43]. This approach only supports the *workpile* model, since it does not take into account any inter-process dependence.

The result, in our opinion, is that contemporary general-purpose schedulers are too focused on satisfying a small set of requirements, while missing the “big picture”. Largely, they overlook the two requirements that we believe are critical for performance and efficiency for parallel desktop workloads: separation of co-interfering processes and coscheduling of collaborating processes.

The growing popularity of parallel programming behooves OS support for all parallel computation modes: from manual coarse grained parallization implemented using thread libraries [21] (using either the BSP or systolic paradigms), to compiler auto-parallelization techniques that are mainly based based on BSP [56, 6, 30, 33, 1].

The main challenge general-purpose OSs are facing when it comes to scheduling both BSP and systolic parallel applications is how to incorporate the inter-thread dependencies in the process scheduling. This dependence information can be explicitly communicated by the user through specific interfaces to the OS, as implemented in contemporary preemptible cluster management systems [11]. In contrast, a more interesting approach is to deduce these dependencies implicitly by tracking interprocess communication at runtime and deriving the resulting scheduling requirements. Several studies has shown the promise in this design, especially in uncovering hidden dependencies [57, 13, 19, 53] — for example, those involving system dæmons that are not an integral part of the application [13].

Let us now explore how these challenges translate to actual mis-scheduling scenarios with case studies in the next section.

### 3 Case Studies

To demonstrate the importance of OS awareness of parallel constraints we discuss several case studies, including both systolic and BSP applications, using different software and hardware variations.

These examples were kept intentionally simple to 1) facilitate reproduction of our results and “benchmarks”; and, 2) reduce the effect of complex, unmodeled relationships between hardware and software components, such as memory hierarchy considerations, I/O performance, etc. We did attempt however to evaluate a wide range of hardware and OS configurations, which are detailed in Table 1.

Name	Processor technology	Operating System	Processors
P3	Pentium-III 664 MHz	Linux 2.4.8	1
P4	Pentium-IV 2800 MHz	Various	1
ES40	HP Alpha EV6 833MHz	Linux 2.4.21	4
ES45	HP Alpha EV6 1250MHz	Tru64 5.1	4
IBM-2.4/2.6	Pentium-III 550MHz	Linux 2.4.22/2.6.9	4
Potomac	Hypertheaded Xeon 3330MHz	Linux 2.6.11	4 (8)

Table 1: Experimental platforms (number in parenthesis is logical processors on SMTs)

### 3.1 Systolic Applications: A Movie Story

The scenario we describe here may be familiar to most readers: running a latency-sensitive application, such as a media player or a Voice-over-IP application with other tasks in the background, only to suffer from skipped frames and poor playback quality. If the root cause of the problem were inadequate computing capacity, we may have been forced to accept lower quality or less multitasking. However the problem results strictly from mis-scheduling, as evidenced by the many attempts to address latency-sensitive applications in the scheduling literature [5, 10, 12, 24, 37, 38, 41]. Paradoxically, simple, low-resource special-purpose hardware such as a portable video player can present a movie smoothly, while watching a movie on a significantly more powerful PC is often jittery when another application is running in the background — anti-virus scanner or a download client for example — and it must be stopped for smooth viewing. Media scheduling has received much attention lately and recent Linux schedulers provide far better support than older versions. We wish to leverage this experience towards parallel desktop scheduling.

**Methodology** Our experiments are based on measuring the performance of Xine — a multithreaded movie player. Xine’s design systolic parallel, with the two most important threads performing the decoding and displaying of frames. Our workload consist on running Xine with increasing background load and analyzing how different Linux schedulers are affected. This workload was chosen to demonstrate how a scheduler incognizant of process dependencies might degrade the throughput of a real-life parallel application. Moreover, Xine is implicitly dependent on the graphical subsystem — specifically the X windows system — adding another implicit stage to the systolic pipeline [45]. We used the Klogger kernel logging framework to log context switches [14] thus exposing CPU consumption patterns, alongside Xine’s internal dropped frames statistics. During the measurements, Xine processed a short MPEG clip, which resides in a memory filesystem to avoid I/O clutter.

#### 3.1.1 Base case: uniprocessor system

To expose the user-visible effects of poor scheduling decisions on movie playing, we reproduce here results performed on a lower-end machine (P3), with a more disruptive background activity. Even though computer performance continues to grow, low-end processors continue to be of interest because of their wide use in portable, power-aware, or embedded environments. Even on high-end machines workloads continue to require increasingly more resources (Section 1.2), therefore increasing the background interference and load. The experiments consist of running Xine with an increasing number of CPU stressors (that iterate on an empty loop) with two Linux schedulers: 2.4.8 and an improved research Linux version (based on Linux 2.4) that automatically identifies and prioritizes interactive processes [12, 13]. Here, we measure the distribution of CPU utilization among processes, as well as the user-noticeable percentage of skipped frames that Xine suffers. The Linux 2.4.8 results, shown in Figure 4(a), indicate that a generic scheduler not respecting the scheduling requirements of a soft-real-time application’s entire systolic thread group (Xine), the increase in background load translates directly to a decrease in interactive responsiveness. Conversely, an interprocess dependency tracking scheduler (Figure 1(b)) prioritize the thread group as a whole and uncover its implicit systolic stage (X server), thus enabling sustained frame rate.

#### 3.1.2 Emerging platform: multiprocessor system

Uniprocessor systems are limited to temporal partitioning, whereas multiprocessors schedulers can also utilize spatial partitioning. To show the inadequacy of the standard *workpile* oriented scheduler to handle systolic application and the potential in spacial partitioning, we repeated the Xine experiment on IBM-2.6. The stressors were modified to load the memory bus and caches by randomly accessing a memory range bigger than the L2 cache.

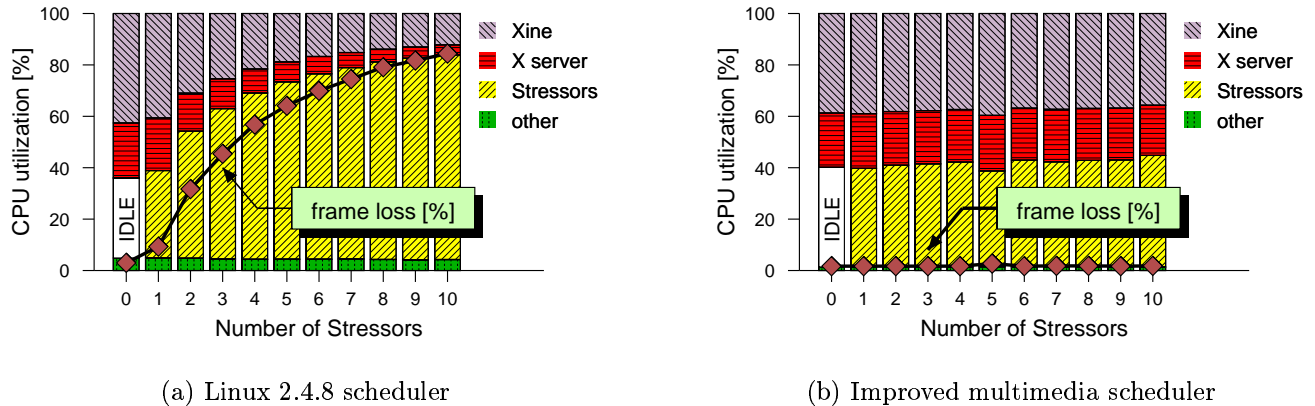


Figure 1: Comparison of two schedulers running Xine under increasing background load on “P3”

Figure 2(a) shows the system unpredictable behavior using the default scheduler, degrading performance as the load increases. Specifically, when running up to two stressors Xine and X get a dedicated CPU each. The temporary performance loss observed when running 3 stressors is caused by the scheduler attempt to balance the load pushing both Xine and X onto the same processor, making them compete against each other. When running 4-7 stressors however, the scheduler cannot balance the load and again separates Xine and X, letting each compete with a different stressor — but with a slight interactive priority gain — causing stressors to effectively replace the idle processes. These unpredictable migrations also explain the inconsistent performance seen throughout the measurement. Conversely, it may even lead to a positive effect where imbalances improve Xine’s performance (10 stressors).

Figure 2(b) shows the results of the same experiment, after isolating Xine and X’s affinity to run on processor 0 exclusively. It shows that while a single processor cannot accommodate the computational needs of both applications, the performance is consistent and not dependent on the increasing load. The only two inconsistencies (when running 12 and 18 background processes) are caused by awakenings of the system swap daemon.

Since both Xine and X are multimedia/interactive applications, we wanted to see if the scheduler can handle balancing only these over time. As such we manually partitioning the machine having Xine and X on processors 0–1 and the stressors on the others. The results are shown in Figure 2(c). We can see that in general this indeed yielded minor frame loss as both Xine and X got an exclusive processor each. However, during the consecutive executions of Xine (11 stressors) the system swap daemon woke up on processor 1 driving out Xine to compete with X on processor 0. Even though the swap daemon ran for a short period, the scheduler only migrated Xine back after a few minutes — leaving processor 1 idle for four consecutive measurements. These effects were verified with repeating measurements.

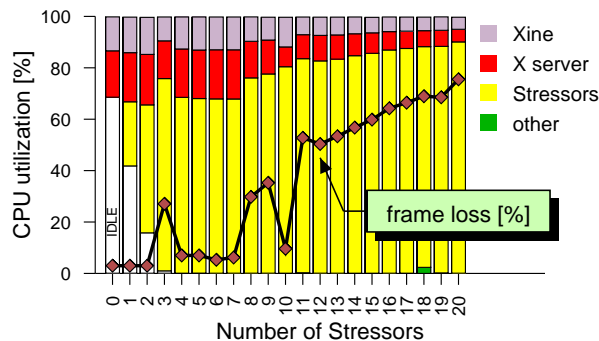
Finally, when manually isolating X on processor 0, Xine on processor 1, and letting the scheduler handle with the stressors on processors 2–3 as depicted in Figure 2(d) we observe that the performance is both predictable and optimal.

### 3.1.3 Summary

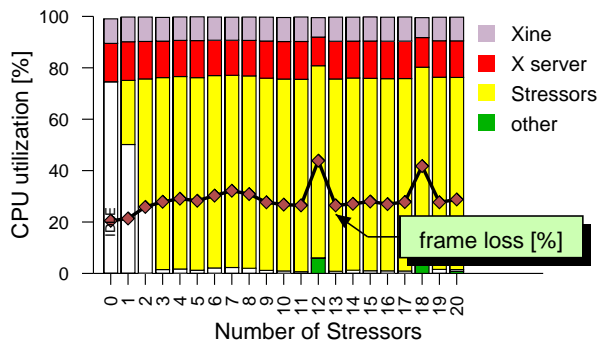
Scheduling on multiprocessors bring forth the conflict of easily collected local knowledge vs. the difficulty to collect global knowledge. In this section we have explored this tradeoff for systolic parallel applications, showing how utilizing only local migration decision on a multiprocessor poorer performance and even worse — it may lead to unpredictable results. Further more, we have shown that migration — when used appropriately — can provide optimal performance.

## 3.2 Bulk Synchronous Applications: Machine Global Scheduling

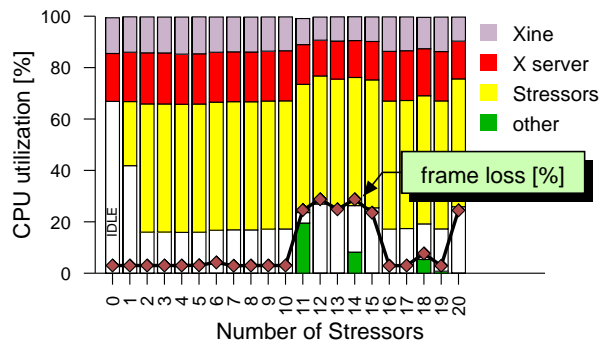
Parallel job scheduling — an active and developed topic in the realm of supercomputing — has not received much attention in the context of commodity machines. Nevertheless, if we are indeed heading toward commodity parallel architectures as described in Section 1, the scheduling requirements of BSP jobs in mixed workloads will have to be addressed. As such, we designed a simple experiment to measure the basic interferences in a mixed workload of sequential and parallel applications.



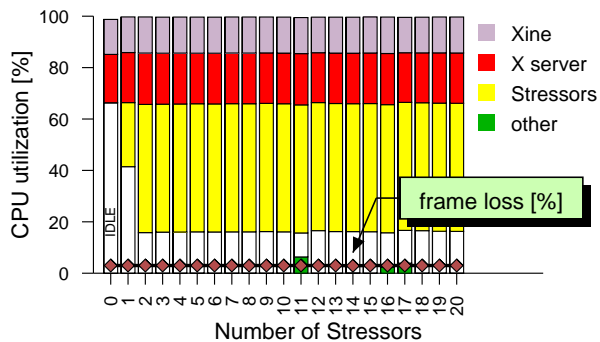
(a) No processor affinity restrictions



(b) Isolating Xine+X on CPUs 0



(c) Isolating Xine+X on CPUs 0-1



(d) X on CPU0, Xine on CPU1

Figure 2: The effect of imposing processor affinity heuristics on the Linux 2.6.9 scheduler running on “IBM-2.6”. The utilization is accumulated over all 4 CPUs.

### 3.2.1 Experiment Description

A wrapper program launches a set of sequential programs (“stressors”) and parallel programs. The stressors consume cycles with simple computation running in an infinite loop. The parallel programs execute the same computation but for a predefined number of iterations. In addition, every few hundreds of iterations they synchronize with each other using standard Unix semaphores. This structure follows the bulk-synchronous parallel (BSP) model, which in practice serves to capture the structure of many real parallel programs [55]. Each parallel program is composed of a number of threads equal to the number of processors less one. The one unallocated processor allows the mitigation of the effect of system daemons and unrelated user-level processes that might wake up periodically, especially when gang scheduling (GS) is concerned [15, 40].<sup>1</sup>

The wrapper program waits for the completion of all the parallel programs before it terminates the sequential programs and the total time to completion is measured. By varying the number of parallel programs and stressors we can measure the performance effect of the host OS scheduler on different workload mixes. In addition, the wrapper program has a gang scheduling mode where time is sliced into slots. On each slot, either one parallel program, or all the sequential programs, are running exclusively (other programs are suspended with a SIGSTOP signal). Thus, parallel programs can enjoy a dedicated view of the machine for the duration of their timeslice, which facilitates their frequent synchronization. Time slices are switched in a round-robin fashion.

To reduce the effect of variability and noisiness in the results, we repeated each run at least five times, discarded the minimum and maximum results, and averaged the remaining run times. By focusing on completion time, this experiment is designed to measure the effect on the parallel jobs, since the sequential applications continue to run for an indefinite amount of time and, representing a relatively constant background load.

<sup>1</sup>A “pure” GS implementation at the OS level would simply set aside a timeslot for housekeeping tasks and low priority processes [51].

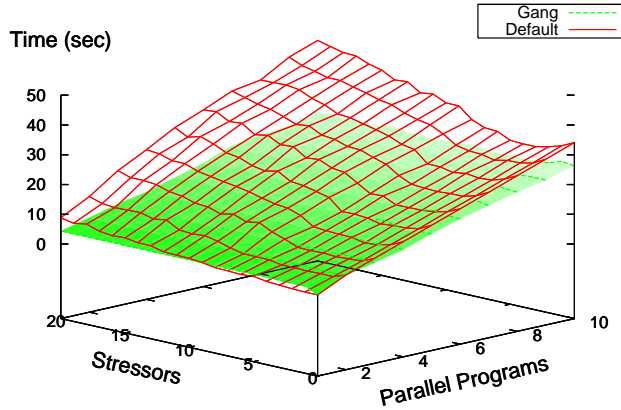


Figure 3: Gang scheduling vs. default OS scheduling on ES40

### 3.2.2 Results and Analysis

We ran this experiment with 1 to 10 parallel programs and 0 to 20 sequential stressors on all the parallel architectures in Table 1. Fig. 3 compares the performance of both schedulers on ES40. As might be expected, the total run time for both gang and default scheduling increases with the number of parallel jobs. On ES40, the default scheduler (Linux 2.4.21) does a poor job of coscheduling the threads of the parallel programs, resulting in higher run times than under GS. The gap in the scheduler performance grows as the load increases, both in terms of parallel and sequential programs. On the opposite corner, running a single parallel program with few stressors yields performance which is slightly worse for GS. The main reason for this is that the single parallel job, being the only one that blocks, gets prioritized in Linux 2.4 over the nonblocking sequential jobs. In addition, the relatively high time quantum of GS (compared to total execution time) and overhead added by the extra scheduling layer contribute to some performance degradation with GS.

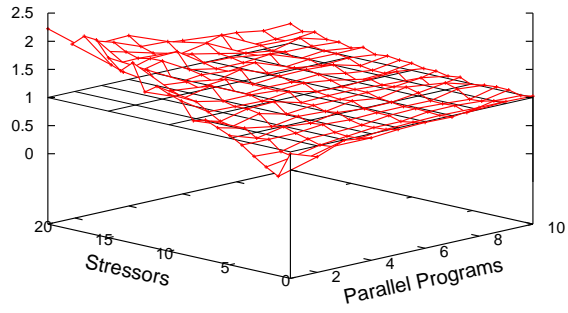
To compare default and gang scheduling across architectures, we can no longer use absolute run times, as those depend on the architecture. We therefore normalize the results by calculating and plotting the slowdown of the default scheduler when compared to GS. The results, shown in Fig. 4, are compared to the unit surface, which represents equal run time of the gang and default scheduler. A slowdown higher than one represents a scenario where gang scheduling performs better than the default scheduler, while the opposite is true for values below the unit surface.

We start by comparing the effect of different OS schedulers on the same architecture (IBM-2.4/2.6 in Figure 4). In the absence of sequential jobs, parallel jobs that block on communication tend to self-synchronize, since a thread that tries to synchronize with a blocked thread will eventually block itself, releasing the CPU for another program that is ready to run[4]. Consequently, explicit gang scheduling is not required to achieve synchronization in a mostly-parallel workload when a blocking mechanism is used (in this case, semaphores). This effect is evident when looking at the Stressors=0 axis of the figures. In fact, the OS timer interrupt frequency (HZ in Linux) in version 2.6 is 10 times finer than in 2.4, resulting in a faster context switch to a newly-unblocked parallel process, and therefore the parallel programs are prioritized over the CPU-bound sequential programs. Indeed, self-synchronization is so effective in 2.6 that the default scheduler significantly outperforms the coarse-grained gang scheduler in low sequential loads. However, as the number of sequential programs increases, and they can no longer be accommodated on the spare processor, synchronization of the parallel programs is interfered, and their runtime grows. This is especially evident on the Parallel=1 axis, where the weight of the single parallel program in the workload diminishes as the number of stressors grows, and by not ensuring the coscheduling of its threads, the Linux scheduler hampers the parallel program's progress.

Another important property of a scheduler is its stability and predictability. The IBM-2.6 figure shows a much smoother surface than that of IBM-2.4. Analyzing the results per data point reveals that the average span of measured values for the Linux 2.4 scheduler is more than triple that of Linux 2.6's scheduler, possibly because of the coarser grain at which scheduling decisions are taken. In addition, the span of measured values for both OSs is larger than that of GS, especially for higher loads. This is a direct consequence of the scheduling order and determinism forced by GS [51].

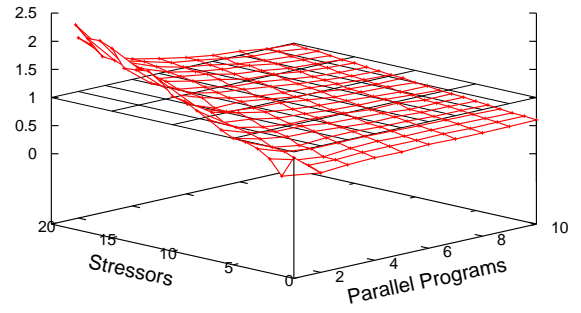
The next pair of figures (ES40 and ES45) again compare two different OS schedulers (Linux 2.4 and Tru64 5.1) on similar architectures. On this architecture, Linux 2.4's scheduler performs even worse than on the IBM, in

Slowdown



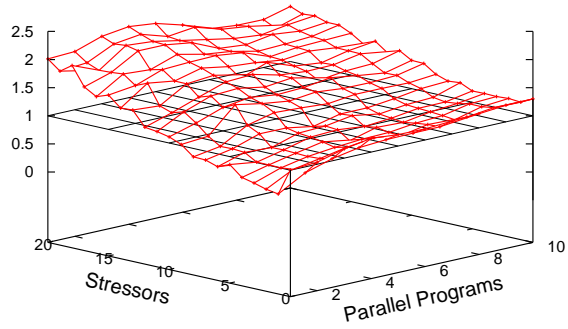
(a) IBM-2.4 (Linux 2.4.22)

Slowdown



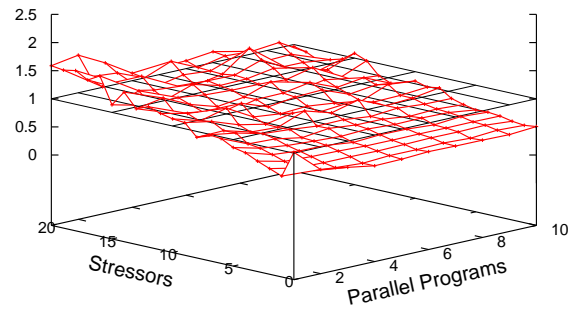
(b) IBM-2.6 (Linux 2.6.9)

Slowdown



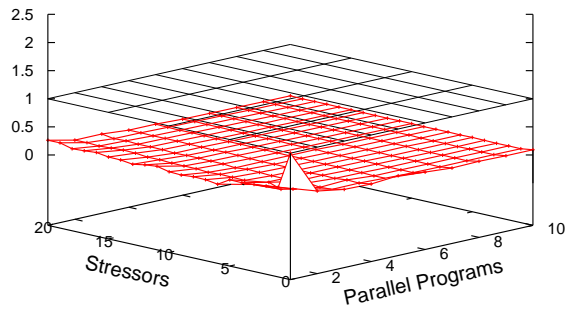
(c) ES40 (2.4.21)

Slowdown



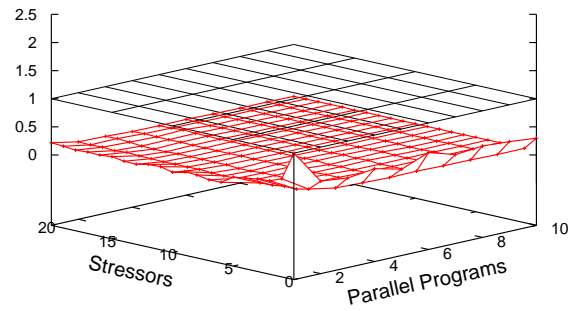
(d) ES45 (Tru64 5.1)

Slowdown



(e) Potomac—4 threads (Linux 2.6.11)

Slowdown



(f) Potomac—7 threads (Linux 2.6.11)

Figure 4: Slowdown comparison of various architectures and OSs

terms of slowdown. The reason for this is that having faster processors, the effective synchronization granularity of the parallel programs is finer on the ES40 than on the IBM, increasing its sensitivity to mis-scheduling and noise [28, 29, 52]. Tru64’s scheduler, designed from the start for multiprocessor servers, does remarkably well with the parallel programs. For most runs with eight stressors or less, Tru64’s scheduler outperforms GS regardless of the number of parallel programs. It does however show significant variability in the results, occasionally taking twice as long to complete the same experiment. We believe this is related to Tru64’s high susceptibility to OS noise when all processors are employed, as shown in a related analysis on a nearly identical architecture [40]. Unfortunately, we do not have access to Tru64’s source code to verify this hypothesis.

In the examples described so far GS is a preferable policy to the *parallel application*, potentially at the cost of sequential and interactive programs<sup>2</sup>. Is this always the case? The last pair of graphs shows that enforcing coscheduling on a partial resource allocation or on an Asymmetric MultiProcessor (AMP) is not always beneficial to the parallel program. Fig 4(e) shows the results of running the same workload (i.e., with 4 threads per parallel program) on the Potomac, which has 8 logical processors (4 hyperthreaded Xeon MPs)<sup>3</sup>. Limiting the number of threads to 4 wastes many compute resources that are otherwise filled by the default scheduler. But resource wasting is not the only problem in an AMP, as shown in Fig. 4(f). Increasing the number of threads per parallel program to 7 again shows poor GS performance. The source of the problem is that every two virtual processors share many of their internal resources, so the threads of each parallel program end up competing with each other instead of complementing each other [49].

### 3.2.3 Summary

In this section we demonstrate one principle surfaces in virtually all the works on parallel job scheduling: the need to coschedule tightly-coordinated tasks, especially in mixed workloads. Our experiments confirm that as load increases and the workload mixes more sequential and parallel jobs, coscheduling offers substantial performance benefits, even in the form of the relatively rigid gang scheduling algorithm [19]. As architectures grow more parallel and heterogeneous, coscheduling can become a boon or a bane also for unrelated, (and unsynchronized) tasks: coscheduling resource-complementing processes can lead to high parallel speedups, while ignoring these considerations can lead to processes competing with each other over shared resources (such as the memory bus or processor cache) [2, 7, 8, 25].

## 4 Conclusions and Discussion of Future Work

As commodity computers and their workloads continue to evolve, the schedulers that manage them grow increasingly inadequate. The aging schedulers’ limitations attract little attention because they are good enough for single-processor computers, and with the move to parallel architectures, we can no longer afford to ignore these limitations. On the other hand, while parallel job scheduling is a challenging research area with many open questions still being actively studied [16] may not suffice, because these techniques were developed for much more homogeneous workloads.

The scheduling of a mixed workload, combining traditional and new desktop applications with parallel jobs, is only now starting to gain attention, in conjunction with the expected prevalence of commodity parallel architectures.

We posit that to be truly general-purpose for current and future workloads, commodity schedulers cannot maintain the same anachronistic principles with which they were conceived 30 years ago (with the occasional ad-hoc provisions added to address specific problems such as multimedia). Our experimental results show that contemporary scheduling methods used on parallel workloads can lead to significant application slowdown, diminished user experience, and even unpredictability. Furthermore, we show how applying common parallel scheduling techniques can go a long way.

However, the picture is not all grim. We believe that adapting techniques from different scheduling domains can yield a unified process scheduler. Two principles underlie all the decisions of our scheduler proposal:

**Maximizing collaboration:** processes that require or benefit from coscheduling should be coscheduled

**Minimizing interference:** processes that have conflicting requirements should be separated in time or space

Note that these principles also implicitly address different architectures and workloads. For example, interference can be the result of scheduling two processes on the same hyperthreaded processor, and assigning one of them

<sup>2</sup>In some cases, coarse-grained GS has benefits for sequential applications as well, due to cache effects [8].

<sup>3</sup>With the widespread availability of dual-core processors, and the near-future release of quad-core processors, Potomac may be considered a good representative of future desktop machines.

to a different processor (spacial partitioning) can benefit both processes. As another example, processes that are collaborating (parallel threads) or codependent (data dependency) could require coscheduling.

In our ongoing research work to develop scheduling schemes we are exploring the following methods to bring these high-level principles to practice:

**Classification:** By gathering runtime data on processes and process groups, the OS can classify them based on its scheduling requirements (see for example other works on process classification [3, 46, 48, 19, 53, 57]).

**Flexible scheduling:** The scheduler can also benefit from dynamically testing different allocations of time and space, while monitoring the progress of processes [49]. Scheduling should be based on progress feedback, as different combinations can have significant performance impact, especially if heterogeneity or energy management also come into play [7, 48].

**Adaptivity:** The scheduler must remain adaptive to adjust to changes in workloads. Adaptivity can be expressed in various forms, such as choice of scheduling policies, parameters, process priorities, and time-slice frequency and length [24, 48].

**Cost functions:** Cost functions are a generic mechanism to change the goals of the scheduler to the specific environment. For example, a scheduler on a low-power device can aim for energy-efficient scheduling [32] or reduce bus contention [2].

We plan to evaluate these ideas with actual experiments on a variety of contemporary workloads and architectures. Looking into the more distant future, scalability issues could warrant even more changes in the scheduler: tens and hundreds of cores may require the OS to effectively be of a distributed nature. Additional challenges might stem from power management heterogeneity considerations. Hopefully, these considerations can also fall under the umbrella of the principles we suggest in this paper.

## References

- [1] A. Aiken and A. Nicolau. Optimal loop parallelization. In *International Conference on Programming Language Design and Implementation*, pages 308–317, New York, NY, USA, 1988. ACM Press.
- [2] Christos D. Antonopoulos, Dimitrios S. Nikolopoulos, and Theodore S. Papatheodorou. Scheduling algorithms with bus bandwidth considerations for SMPs. In *32nd International Conference on Parallel Processing (ICPP)*, Kaohsiung, Taiwan, October 2003. Available from [www.cs.wm.edu/~dsn/papers/icpp03.pdf](http://www.cs.wm.edu/~dsn/papers/icpp03.pdf).
- [3] Remzi H. Arpaci, Andrea C. Dusseau, Amin M. Vahdat, Lok T. Liu, Thomas E. Anderson, and David A. Patterson. The interaction of parallel and sequential workloads on a network of workstations. In *SIGMETRICS Measurement & Modeling of Computer Systems*, pages 267–278, Ottawa, Canada, May 1995. Available from [citeseer.ist.psu.edu/arpaci95interaction.html](http://citeseer.ist.psu.edu/arpaci95interaction.html).
- [4] Andrea C. Arpaci-Dusseau. Implicit Coscheduling: Coordinated scheduling with implicit information in distributed systems. *ACM Transactions on Computer Systems*, 19(3):283–331, August 2001. Available from [portal.acm.org/ft\\_gateway.cfm?id=380764&type=pdf](http://portal.acm.org/ft_gateway.cfm?id=380764&type=pdf).
- [5] Scott A. Banachowski and Scott A. Brandt. The BEST Scheduler for Integrated Processing of Best-Effort and Soft Real-Time Processes. In *Multimedia Computing and Networking (MMCN)*, San Jose, CA, January 2002. Available from [www.cse.ucsc.edu/~sbanacho/papers/banachowski-mmcn02.ps](http://www.cse.ucsc.edu/~sbanacho/papers/banachowski-mmcn02.ps).
- [6] Rob H. Bisseling. *Parallel Scientific Computation A Structured Approach using BSP and MPI*. Oxford University Press, Mar 2004.
- [7] James R. Bulpin and Ian A. Pratt. Multiprogramming performance of the Pentium 4 with hyper-threading. In *Second Annual Workshop on Duplicating, Deconstruction and Debunking (WDDD)*, pages 53–62, Munchen, Germany, June 2004. Available from [www.ece.wisc.edu/~wddd/2004/06\\_bulpin.pdf](http://www.ece.wisc.edu/~wddd/2004/06_bulpin.pdf).
- [8] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *14th ACM Symposium on Operating Systems Principles (SOSP)*, pages 120–133, Asheville, NC, December 1993. ACM Press. Available from [citeseer.ist.psu.edu/chen93impact.html](http://citeseer.ist.psu.edu/chen93impact.html).
- [9] Budi Darmawan, Charles Kamers, Hennie Pienaar, and Janet Shiu. *AIX 5L Performance Tools Handbook*. IBM Redbooks, 2nd edition, Aug 2003.

- [10] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (BVT) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler. In *17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 261–276, Charleston, SC, December 1999. Available from [citeseer.ist.psu.edu/duda99borrowedvirtualtime.html](http://citeseer.ist.psu.edu/duda99borrowedvirtualtime.html).
- [11] Yoav Etsion and Dan Tsafir. A short survey of commercial cluster batch schedulers. Technical Report 2005-13, Hebrew University, May 2005.
- [12] Yoav Etsion, Dan Tsafir, and Dror G. Feitelson. Desktop scheduling: How can we know what the user wants? In *14th ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, pages 110–115, County Cork, Ireland, June 2004. Available from [www.cs.huji.ac.il/~feit/papers/HuCpri04NOSSDAV.pdf](http://www.cs.huji.ac.il/~feit/papers/HuCpri04NOSSDAV.pdf).
- [13] Yoav Etsion, Dan Tsafir, and Dror G. Feitelson. Process prioritization using output production: Scheduling for multimedia. *ACM Transactions on Multimedia Computing, Communications, and Applications*, 2(4), 2006. to appear.
- [14] Yoav Etsion, Dan Tsafir, Scott Kirkpatrick, and Dror G. Feitelson. Fine grained kernel logging with klogger: Experience and insights. Technical Report 2005-35, School of Computer Science and Engineering, Hebrew University, June 2005.
- [15] Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, December 1992. Available from [www.cs.huji.ac.il/~feit/papers/GangPerf92JPDC.ps.gz](http://www.cs.huji.ac.il/~feit/papers/GangPerf92JPDC.ps.gz).
- [16] Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Parallel job scheduling – A status report. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Tenth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 3277 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2004. Available from [www.cs.huji.ac.il/~feit/parsched/](http://www.cs.huji.ac.il/~feit/parsched/).
- [17] Krisztián Flautner, Rich Uhlig, Steve Reinhardt, and Trevor Mudge. Thread-level parallelism of desktop applications. In *Workshop on Multi-Threaded Execution, Architecture, and Compilers (MTEAC)*, Toulouse, France, January 2000. Available from [www-cse.ucsd.edu/users/tullsen/mteac2000/flautner.pdf.gz](http://www-cse.ucsd.edu/users/tullsen/mteac2000/flautner.pdf.gz).
- [18] Michael J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, Dec 1966.
- [19] Eitan Frachtenberg, Dror G. Feitelson, Fabrizio Petrini, and Juan Fernandez. Flexible CoScheduling: Mitigating load imbalance and improving utilization of heterogeneous resources. In *17th International Parallel and Distributed Processing Symposium (IPDPS)*, Nice, France, April 2003. Available from [www.cs.huji.ac.il/~etcs/pubs/](http://www.cs.huji.ac.il/~etcs/pubs/).
- [20] Bjorn Franke and Michael F.P. O’Boyle. A complete compiler approach to auto-parallelizing c programs for multi-dsp systems. *IEEE Transactions on Parallel and Distributed Systems*, 16(3):234–245, Mar 2005.
- [21] Felix Garcia and Javier Fernandez. POSIX thread libraries. *Linux Journal*, Feb 2000.
- [22] W. Wayt Gibbs. A split at the core. *Scientific American*, 291(5):96–101, November 2004. Available from [www.sciam.com/article.cfm?articleID=00026625-6DF0-1179-ADF083414B7FFE9F](http://www.sciam.com/article.cfm?articleID=00026625-6DF0-1179-ADF083414B7FFE9F).
- [23] Milind Girkar and Constantine D. Polychronopoulos. Extracting task-level parallelism. *ACM Transactions on Programming Languages and Systems*, 17(4):600–634, 1995.
- [24] Ashvin Goel, Luca Abeni, Charles Krasic, Jim Snow, and Jonathan Walpole. Supporting time-sensitive applications on a commodity OS. In *Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–180, Boston, MA, December 2002. Available from [citeseer.ist.psu.edu/goel02supporting.html](http://citeseer.ist.psu.edu/goel02supporting.html).
- [25] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *SIGMETRICS Measurement & Modeling of Computer Systems*, pages 120–32, San Diego, CA, May 1991. Available from [discolab.rutgers.edu/classes/cs519-old/papers/p120-gupta.pdf](http://discolab.rutgers.edu/classes/cs519-old/papers/p120-gupta.pdf).

- [26] Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. Detecting coarse - grain parallelism using an interprocedural parallelizing compiler. In *IEEE/ACM Supercomputing*, page 49, Washington, DC, USA, 1995. IEEE Computer Society. Available from <http://doi.ieeecomputersociety.org/10.1109/SUPERC.1995.25>.
- [27] H. Peter Hofstee. Power efficient processor architecture and the Cell processor. In *11th International Symposium on High-Performance Computer Architecture*, San Francisco, CA, February 2005. Available from [www.hpcaconf.org/hpca11/papers/25\\_hofstee-cellprocessor\\_final.pdf](http://www.hpcaconf.org/hpca11/papers/25_hofstee-cellprocessor_final.pdf).
- [28] James Patton Jones and Bill Nitzberg. Scheduling for parallel supercomputing: A historical perspective of achievable utilization. In Dror G. Feitelson and Larry Rudolph, editors, *Fifth Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1659 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1999. Available from [www.cs.huji.ac.il/~feit/parsched/](http://www.cs.huji.ac.il/~feit/parsched/).
- [29] Terry Jones, William Tuel, Larry Brenner, Jeff Fier, Patrick Caffrey, Shawn Dawson, Rob Neely, Robert Blackmore, Brian Maskell, Paul Tomlinson, and Mark Roberts. Improving the scalability of parallel jobs by adding parallel awareness to the operating system. In *15th IEEE/ACM Supercomputing*, Phoenix, AZ, November 2003. ACM Press and IEEE Computer Society Press. Available from [www.sc-conference.org/sc2003/paperpdfs/pap136.pdf](http://www.sc-conference.org/sc2003/paperpdfs/pap136.pdf).
- [30] Ken Kennedy and Randy Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, Oct 2001.
- [31] Rakesh Kumar, Keith Farkas, Norman Jouppi, Partha Ranganathan, and Dean Tullsen. A multi-core approach to addressing the energy-complexity problem in microprocessors. In *Workshop on Complexity-Effective Design (WCED)*, June 2003. Available from [citeseer.ist.psu.edu/kumar03multicore.html](http://citeseer.ist.psu.edu/kumar03multicore.html).
- [32] Yingmin Li, David Brooks, Zhigang Hu, and Kevin Skadron. Performance, energy, and thermal considerations for SMT and CMP architectures. In *11th International Symposium on High-Performance Computer Architecture*, San Francisco, CA, February 2005. Available from [www.hpcaconf.org/hpca11/papers/08\\_x\\_li\\_smtandcmparchitectures.pdf](http://www.hpcaconf.org/hpca11/papers/08_x_li_smtandcmparchitectures.pdf).
- [33] David J. Lilja. Exploiting the parallelism available in loops. *Computer*, 27(2):13–26, 1994.
- [34] Robert Love. *Linux Kernel Development*. Novell Press, Indianapolis, IN, USA, 2nd edition, 2005.
- [35] Marshall Kirk McKusick and George V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. pub-AW, Aug 2004.
- [36] Jason Nieh, James G. Hanko, J. Duane Northcutt, and Gerard A. Wall. SVR4 UNIX scheduler unacceptable for multimedia applications. In *Fourth ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, November 1993. Available from [citeseer.ist.psu.edu/443381.html](http://citeseer.ist.psu.edu/443381.html).
- [37] Jason Nieh and Monica S. Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 184–197, Saint-Malo, France, October 1997. Available from [citeseer.ist.psu.edu/547416.html](http://citeseer.ist.psu.edu/547416.html).
- [38] Jason Nieh and Monica S. Lam. Multimedia on multiprocessors: Where’s the OS when you really need it? In *Eighth ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, June 1998. Available from [citeseer.ist.psu.edu/430008.html](http://citeseer.ist.psu.edu/430008.html).
- [39] John K. Ousterhout. Why aren’t operating systems getting faster as fast as hardware? In *Summer 1990 USENIX Conference*, pages 247–256, Anaheim, CA, June 1990. USENIX Association, IEEE Press. Available from [citeseer.ist.psu.edu/165263.html](http://citeseer.ist.psu.edu/165263.html).
- [40] Fabrizio Petrini, Darren Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *15th IEEE/ACM Supercomputing*, Phoenix, AZ, November 2003. Available from [www.c3.lanl.gov/~fabrizio/papers/sc03\\_noise.pdf](http://www.c3.lanl.gov/~fabrizio/papers/sc03_noise.pdf).
- [41] Melissa A. Rau and Evgenia Smirni. Adaptive CPU scheduling policies for mixed multimedia and best-effort workloads. In *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 252–261, College Park, MD, October 1999. Available from [citeseer.ist.psu.edu/rau99adaptive.html](http://citeseer.ist.psu.edu/rau99adaptive.html).

- [42] Mendel Rosenblum, Edouard Bugnion, Herrod Herrod, Emmett Witchel, and Anoop Gupta. The impact of architectural trends on operating system performance. In *15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 285–298, Copper Mountain, CO, December 1995. ACM SIGOPS. Available from [www.ugrad.cs.jhu.edu/~cs318/papers/archtrends-sosp95.ps](http://www.ugrad.cs.jhu.edu/~cs318/papers/archtrends-sosp95.ps).
- [43] Mark E. Russinovich and David A. Solomon. *Microsoft Windows Internals*. Microsoft Press, 4th edition, Dec 2004.
- [44] Stefan Rusu. Trends and challenges in high-performance microprocessor design. Presentation available from [www.eda.org/edps/edp04/submissions/presentationRusu.pdf](http://www.eda.org/edps/edp04/submissions/presentationRusu.pdf), April 2004.
- [45] Robert W. Scheifler and Jim Gettys. The X Window system. *ACM Transactions on Graphics*, 5(2):79–109, 1986.
- [46] Margo Seltzer and Christopher Small. Self-monitoring and self-adapting operating systems. In *Sixth Workshop on Hot Topics in Operating Systems (HotOS)*, pages 124–129, Cape Cod, MA, May 1997. Available from [www.eecs.harvard.edu/vino/vino/papers/monitor.ps](http://www.eecs.harvard.edu/vino/vino/papers/monitor.ps).
- [47] Stephen Shankland. Intel’s dual-core Xeon due in 2006. [news.com.com/2100-1006\\_3-5416330.html](http://news.com.com/2100-1006_3-5416330.html), October 2004.
- [48] Allan Snaveley, Dean Tullsen, and Geoff Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *SIGMETRICS Measurement & Modeling of Computer Systems*, pages 66–76, Marina Del Rey, CA, June 2002. Available from [citeseer.ist.psu.edu/528307.html](http://citeseer.ist.psu.edu/528307.html).
- [49] Allan Snaveley and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 234–244, Cambridge, MA, November 2000. Available from [citeseer.ist.psu.edu/338334.html](http://citeseer.ist.psu.edu/338334.html).
- [50] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs’s Journal*, 30(3), March 2005. Available from [www.gotw.ca/publications/concurrency-ddj.htm](http://www.gotw.ca/publications/concurrency-ddj.htm).
- [51] Paul Terry, Amar Shan, and Pentti Huttunen. Improving application performance on HPC systems with process synchronization. *Linux Journal*, (127):68–73, November 2004. Available from [www.linuxjournal.com/article/7690](http://www.linuxjournal.com/article/7690).
- [52] Dan Tsafir, Yoav Etsion, Dror G. Feitelson, and Scott Kirkpatrick. System noise, OS clock ticks, and fine-grained parallel applications. In *19th ACM International Conference on Supercomputing*, pages 302–312, Boston, MA, June 2005. Available from [www.cs.huji.ac.il/~feit/papers/Noise05ICS.pdf](http://www.cs.huji.ac.il/~feit/papers/Noise05ICS.pdf).
- [53] Dan Tsafir and Dror G. Feitelson. Barrier synchronization on a loaded SMP using two-phase waiting algorithms. In *16th International Parallel and Distributed Processing Symposium (IPDPS)*, April 2002. Available from [www.cs.huji.ac.il/~feit/papers/Barrier02IPDPS.ps.gz](http://www.cs.huji.ac.il/~feit/papers/Barrier02IPDPS.ps.gz).
- [54] Nathan Tuck and Dean M. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *12th International Conference on Parallel Architecture and Compiler Techniques (PACT)*, pages 26–34, New Orleans, LA, September 2003. IEEE Computer Society. Available from [ieeexplore.ieee.org/iel5/8771/27774/01237999.pdf](http://ieeexplore.ieee.org/iel5/8771/27774/01237999.pdf).
- [55] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990. Available from [portal.acm.org/citation.cfm?id=79181&dl=ACM&coll=portal](http://portal.acm.org/citation.cfm?id=79181&dl=ACM&coll=portal).
- [56] Michael Joseph Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, MA, USA, 1990.
- [57] Haoqiang Zheng and Jason Nieh. SWAP: A scheduler with automatic process dependency detection. In *First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, pages 183–196, San Francisco, CA, March 2004. Available from [www.ncl.cs.columbia.edu/publications/nsdi2004\\_fordist.pdf](http://www.ncl.cs.columbia.edu/publications/nsdi2004_fordist.pdf).
- [58] Albert Zomaya, editor. *Parallel and Distributed Computing Handbook*. Mcgraw-Hill, 1996.